Seminar: XML und Datenbanken
XML-Verarbeitungskonzepte

Sebastian Paul Potthoff
Betreuer: Jürgen Göres

Technische Universität Kaiserslautern
Fachbereich Informatik
Lehrgebiet Informationssysteme

Zusammenfassung Durch die starke Verbreitung von XML als Austauschformat für semistrukturierte Daten wurden verschiedene Verarbeitungskonzepte für den Umgang mit XML-Daten entwickelt. Diese Arbeit stellt drei verschiedene Parserkonzepte in Form von Pull- und Push-Parsern sowie Ein- und Mehrschrittparsern am Beispiel von SAX, DOM und StAX vor. Darüberhinaus werden die XML-Anfragesprachen XQuery/XPath und Xcerpt als Beispiel für navigierende beziehungsweise musterbasierte Anfragesprachen eingeführt.

1 Einführung

1998 wurde XML (Extensible Markup Language) durch das World Wide Web Consortium (W3C)[1] standardisiert. XML beschreibt eine Klasse von Datenobjekten, die als XML-Dokumente bezeichnet werden. Die XML-Spezifikation beschreibt eine Metasprache mit der Sprachen in Form von Dokumenttyp-Definitionen (DTD) oder XML Schema definiert werden. Bekannte Vertreter sind die Sprachen HTML und XHTML [1].

Im allgemeinen Sprachgebrauch wird ein XML-Dokument oft vereinfacht mit der bekannten textuellen Darstellung gleichgesetzt. In ihr werden die Daten mit Hilfe von sogenannten Tags beschrieben. Als Tag wird ein durch spitze Klammern (<, >) eingeschlossener Bezeichner bezeichnet.

Tags dienen der Auszeichnung von Dokumentteilen, die zur Strukturierung der Daten auch geschachtelt werden können. Der auszuzeichnende Teil wird dazu von einem öffnenden (<TagName>) und einem schließenden Tag (</TagName>) umschlossen. Der auf diese Weise gekennzeichnete Bereich wird als *Element* bezeichnet. In XHTML werden beispielsweise Überschriften 1. Ordnung durch umschließende h1-Tags (<h1>Titel</h1>) kenntlich gemacht. Tags können aber auch *leer* (<TagName/>) sein, also keinen Dokumentteil umfassen. Beispielsweise wird ein Zeilenumbruch in XHTML durch das leere
-Tag beschrieben. Darüber hinaus können öffnende Tags um Attribute, also Name-Wert-Paare, erweitert werden (<name alter='23'>). Die Attributwerte werden durch einfache oder doppelte Anführungszeichen abgegrenzt. Es ist nicht möglich Attribute durch weitere Tags zu verfeinern.

Die XML-Spezifikation selbst definiert keine Tags und Attribute. Sie beschreibt lediglich den allgemeinen Aufbau eines XML-Dokuments und die zur Strukturierung verfügbaren Konzepte. So muss ein XML-Dokument beispielsweise *wohlgeformt* sein. Das bedeutet, dass sämtliche XML-Regeln eingehalten werden. Darunter fallen die Existenz genau eines Wurzelelements, das Vorhandensein von öffnenden und schließenden Tags für nicht-leere Elemente sowie einen korrekten Abschluss von leeren Tags. Weiterhin müssen die Tags ebenentreu-paarig geschachtelt sein und Attributbezeichnungen innerhalb eines Elements eindeutig sein.

Unabhängig von der textuellen Darstellung lässt sich eine abstrakte Sicht auf die Daten eines XML-Dokuments definieren, die als XML-Infoset [3] bekannt ist. Das XML-Infoset beschreibt die Daten eines XML-Dokuments als Baumstruktur, die aus verschiedenen Informationseinheiten besteht.

Das Infoset eines wohlgeformten XML-Dokuments enthält eine Informationseinheit für das Dokument als Dokumentwurzel. Von ihm aus sind alle weiteren Informationseinheiten zugänglich. Dazu gehören Informationen über das Dokument sowie eine geordnete Liste aller Kinder des Dokumentelements, Processing Instruction- und Kommentareinheiten.

[1] http://www.w3.org

Jedes der *Elemente* des XML-Dokuments ist durch eine *Element*-Informationseinheit (Element Information Item) repräsentiert. Diese Elemente sind, gemäß ihrer Schachtelung, über die Kindrelation erreichbar. Eine Elementeinheit ist durch ihren lokalen Namen, den Namensraum, die Liste der Kindelemente sowie einer ungeordneten Liste von Attributeinheiten definiert. Weiterhin enthält ein Elementknoten eine Zeichenkette, die den textuellen Inhalt darstellt. Außerdem kann eine Elementeinheit Kommentar- und Processing-Instruction-Einheiten enthalten. Ein *Attribut* innerhalb eines Elements enthält den Namen des Attributs sowie seinen Wert. Der Attributname besteht aus lokalem Namen, dem Namen des Namensraums und wenn vorhanden dem Namensraumpräfix. Die Informationseinheit zu den *Processing Instructions* hat als Eigenschaft die Zielanwendung und den Inhalt der Anweisung. Weiterhin umfasst die Verarbeitungsanweisung einen Verweis auf das Elternelement. Zu jedem Zeichen im Dokument gibt es eine Informationseinheit (Character Information Item). Sie enthält die Unicode-Zeichenkodierung des Zeichens, eine Eigenschaft, die beschreibt, ob Leerzeichen enthalten sind und eine Information zu welchem Element die Zeichendaten gehören. Logisch entspricht jede Informationseinheit einem Zeichen. Den XML-Anwendungen ist jedoch freigestellt mehrere Zeichen zu gruppieren. Die Informationseinheiten für *Kommentare* (Comment Information Items) haben als Eigenschaften das Elternelement sowie den Kommentar selbst. Für jedes Element, für das ein bestimmter Namensraum gültig ist, gibt es eine Informationseinheit (Namespace Information Item).

Diese Arbeit stellt verschiedene Verarbeitungskonzepte für den Zugriff auf derart definierte Daten vor. Zu Beginn werden drei verschiedene Ansätze zur Realisierung von XML-Parsern am Beispiel von SAX, DOM und StAX eingeführt. Der zweite Teil dieser Arbeit beschäftigt sich mit XML-Anfragesprachen, die vor allem im datenbanknahen Umfeld von Bedeutung sind. Hier werden XPath und XQuery, das auf XPath aufbaut, vorgestellt. Bei diesen Sprachen handelt es sich um Sprachen, die sich praktisch als Standard durchgesetzt haben. Darüber hinaus folgt eine Vorstellung der Sprache Xcerpt, die im Vergleich zu XQuery/XPath vollkommen andere Konzepte verfolgt.

2 Parser APIs

Ein Parser ist ein Programm, das als Eingabe ein Dokument erhält und die enthaltenen Daten für die weitere Verarbeitung durch ein Anwendungsprogramm in eine Datenstruktur überführt, auf die programmatisch zugegriffen werden kann. In der XML-Verarbeitung haben sich im wesentlichen zwei Konzepte von Parsern durchgesetzt, Pull- und Push-Parser. Pull-Parser überlassen es der Applikation den Vorgang des Parsens zu kontrollieren. Die Applikation muss jedes Ergebnisse des Parsens explizit anfordern. Push-Parser hingegen besitzen die Kontrolle über die Verarbeitung. Jede vollständig analysierte syntaktische Einheit löst ein Ereignis aus und führt zu einer Benachrichtigung der Applikation (vgl. Abb. 1). Wird das Parsen in einem Schritt vollzogen, spricht man von Einschritt-Parsern.

Ihnen stehen die Mehrschrittparser gegenüber, die einzelne Einheiten Schritt für Schritt analysieren [4].

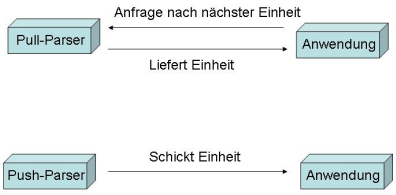

Abbildung 1. Pull- und Push Parser [4]

Im Folgenden werden die Parser SAX, DOM und StAX anhand dieser Kriterien klassifiziert und kurz vorgestellt. SAX und DOM sind als etablierte Parser APIs weit verbreitet und stehen in zahlreichen Implementierungen zur Verfügung. Ein neuerer Ansatz ist StAX.

2.1 SAX

SAX („Simple API for XML“) [5] wurde unter Zusammenarbeit von Peter Murray-Rust, Tim Bray und David Megginson entwickelt. Bis zum Beginn des SAX-Projekts arbeiteten diese drei Entwickler an eigenen Entwicklungen eines XML-Parsers. Peter Murray-Rust initiierte die Diskussion bezüglich einer standardisierten ereignisbasierten API für XML-Parser. Unter Einbeziehung der Mailing-Liste „XML-DEV“ wurde bis Mai 1998 die Version 1.0 von SAX entwickelt. Im Jahr 2000 folgte die Version 2.0. Sie erweiterte SAX um die Fähigkeit XML-Namensräume verarbeiten zu können. Inzwischen wird SAX als SourceForge-Projekt[2] weiterentwickelt.

SAX wurde ursprünglich für Java-Umgebungen entwickelt. Inzwischen existieren jedoch Implementierungen in vielen anderen Programmiersprachen.

Funktionsweise SAX ist ein ereignisorientierter Mehrschritt-Push-Parser. Der Parser arbeitet das Quelldokument sequentiell ab. Während der Verarbeitung erzeugt der Parser eine Sequenz von Ereignissen. Zu den Ereignissen zählen der

[2] http://sourceforge.net

Anfang oder das Ende eines Dokuments (startDocument, endDocument), der Beginn und das Ende eines Elements (startElement, endElement) sowie jede weitere syntaktische Struktur in XML. Der Parser ruft zu jedem Ereignis so genannte Callback-Methoden auf (Push-Parser). Sie sind Bestandteil des Content Handlers, der von der Applikation implementiert wird und die Verarbeitung der Daten übernimmt. Abbildung 2 hebt den Zusammenhang zwischen XML Dokument, Parser und Content Handler noch einmal hervor.

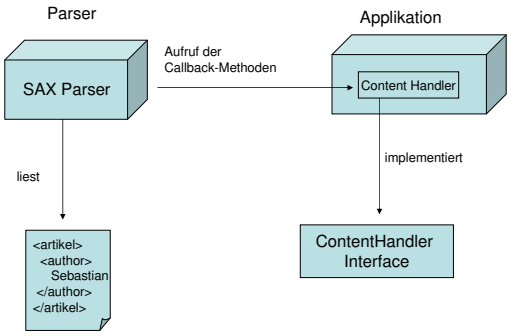

Abbildung 2. Zusammenhang zwischen Quelldokument, Parser und ContentHandler

Bewertung Die Verarbeitung der ereignisspezifischen Informationen benötigt nur wenig Speicherplatz, da das Quelldokument nicht im Hauptspeicher gehalten werden muss. Allerdings werden einmal verarbeitete Informationen nicht durch den Parser gespeichert und sind daher auch nicht erneut erreichbar. Ist ein erneuter Zugriff nötig, muss die Applikation diese selbst speichern. Diese Eigenschaft macht es möglich auch große XML-Dokumente mit SAX-Parsern ressourceneffizient zu verarbeiten. Insbesondere im Kontext mobiler Geräte mit wenig Speicherkapazitäten spricht dieser Punkt für den Einsatz von SAX. Im Vergleich zu DOM-Parsern (vgl. Kap. 1.2) definiert SAX kein Objektmodell [4]. Soll ein eigenes Modell erstellt werden, bietet diese Variante den Vorteil, dass der Mehraufwand eines zweiten, durch die API definierten, Modells entfällt.

Die SAX-API ermöglicht ausschließlich lesenden Zugriff auf XML-Dokumente, und eignet sich daher nicht für Applikationen, die XML-Dokumente erstellen oder modifizieren.

2.2 DOM

Das Document Object Model (DOM) [6] wurde 1998 von der W3C spezifiziert. Wie auch SAX stellt die DOM-Spezifikation nur eine Definition der Schnittstellen zur Verfügung, die von einem konkreten DOM-Parser implementiert werden müssen. Im Gegensatz zu SAX erzeugt der DOM-Parser aus dem XML-Dokument ein Objektmodell im Speicher. Die DOM API definiert, welche Operationen für den Zugriff auf dieses Modell zur Verfügung stehen [4].

Datenmodell Das Objektmodell steht im Mittelpunkt der Arbeit mit DOM. Das Datenmodell beschreibt die XML-Daten als Knotenhierarchie in Form eines Baums. Damit ähnelt das Datenmodell der Struktur des XML-Infosets. Tatsächlich können DOM und SAX als zwei mögliche Umsetzungen des Infosets bezeichnet werden [2].

Die Terminologie des DOM weicht leicht von den Begriffen im XML Infoset ab. Den Informationseinheiten stehen im Document Object Model Knoten gegenüber. Eine Informationseinheit für Kommentare wird somit als Kommentarknoten bezeichnet [3].

Abbildung 3 illustriert wie ein einfacher DOM-Baum für einen Buchshop aussehen kann. Das Dokument verfügt über einen Dokumentwurzelknoten (`document node`), dem der Elementwurzelknoten *buchshop* folgt. Der buchshop-Knoten hat die zwei Unterknoten *buch* und *zeitschrift*. Ein Buch besteht wiederum aus Titel und Author, die jeweils die Daten in Textform enthalten. Darüberhinaus besitzt der buch-Knoten ein Attribut *preis*, das den Wert 10€ hat. Der zeitschrift-Knoten besitzt nur einen Knoten *titel*, der den Titel 'Java' in Textform enthält.

Funktionsweise Der DOM-Parser ist ein Einschritt-Pull-Parser. Er liest das gesamte Dokument ein und erzeugt in einem Schritt ein vollständiges Objektmodell. Mit Hilfe der DOM API bestimmt der Entwickler, wann und wie auf diesen Daten zugegriffen wird. Die Kontrolle liegt somit beim Anwendungsentwickler. Die Navigation im Datenmodell ist anhand der in Abbildung 4 dargestellen Achsen möglich. Der aktuelle Knoten ist der Kontextknoten, von dem aus über die *parent*-Achse der Vater erreicht wird. Der Vaterknoten ist dabei der Knoten, der in einer Preorder Traversierung vor dem Kontextknoten steht. Die Achsen *previousSibling* und *nextSibling* wählen entsprechend die Knoten aus, die in Dokumentreihenfolge direkt vor beziehungsweise nach dem Kontextknoten kommen und den selben Vaterknoten haben. Die einzigen Kindknoten, die direkt über Standardachsen erreicht werden, sind der erste und letzte Kindknoten (firstChild, lastChild). Alle Kindknoten werden zu Knotenlisten (*childNodes*) zusammengefasst und können auf diesem Wege erreicht werden. Neben der

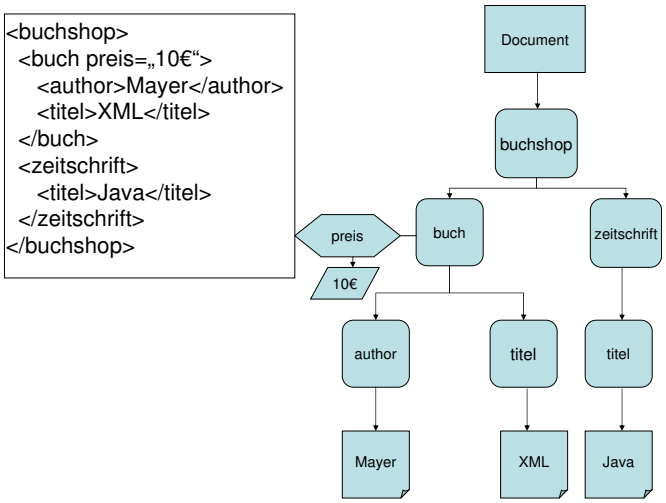

Abbildung 3. Beispiel eines DOM-Baums

Navigation über die vorgestellten Achsen ist ein Zugriff auf die Elemente und Attribute auch anhand ihrer Namen beziehungsweise Objektidentifikatioren (ID) möglich (*getElementById(), getElementByTagName(), getAttribute()*).

Eine weitere wichtige Eigenschaft von DOM ist die Fähigkeit, Änderungen an den XML-Daten vornehmen zu können und sie in ein XML-Dokument auszuschreiben [4].

Bewertung Der Parser legt das erzeugte Objektmodell vollständig im Speicher ab. In Folge dessen ist die Verarbeitung von großen Dokumenten problematisch beziehungsweise ineffizient. Durch den Verbleib des Objektmodells im Speicher, ist ein wiederholter Zugriff auf die Elemente des Modells jederzeit möglich. Weiterhin sind Änderungen am Objektmodell realisierbar.

2.3 StAX

Unter dem JSR (Java Specification Request) 173 'Streaming API for XML', kurz StAX, wurde ein weiterer Ansatz zum Parsen von XML-Dokumenten verfolgt. Ereignisorientierte Parser sind zumeist als Push-Parser umgesetzt worden (vgl. Kap. 1.1). Ziel der Entwicklung von StAX war es, einen ereignisorientierten Pull-Parser zu entwickeln. Die Entwicklung führte das Projekt XMLPull[3] fort, aus dem bereits 2000 eine erste Version der XML Pull API veröffentlicht wurde. Sie verband bereits parallel verlaufenden Entwicklungen (u.a. kXML, XPP) miteinander [7]. 2002 mündete die Entwicklung in der Gründung der Forschungsgruppe

[3] http://www.xmlpull.org

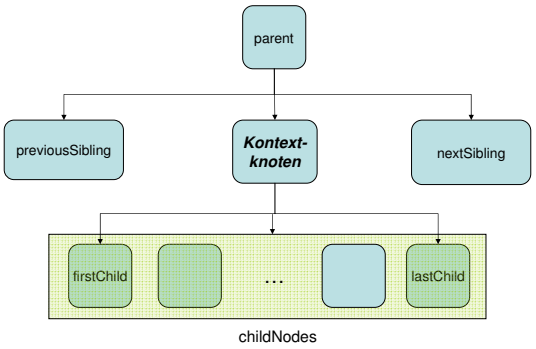

Abbildung 4. Navigationsachsen in DOM

zum JSR 173, die die vorhandene XML Pull Parser in einem Standard zusammenfasste. Die erste vollständige Version wurde am 25. März 2004 veröffentlicht. Seit Java 6 ist StAX Teil der Java-Standardbibliothek.

Arbeitsweise Die Entwicklung eines weiteren Parserkonzepts hatte zum Ziel, die Vorteile der beiden wichtigsten Parser-APIs, SAX und DOM, zu vereinen. Der entscheidene Vorteil von SAX ist die Effizienz, mit der auch auf großen XML-Dokumenten gearbeitet werden kann und relativ unkompliziert eigene Datenmodelle erzeugt werden können. Die Vorteile von DOM wiederum liegen in der Fähigkeit, XML-Dokumente erzeugen zu können. Außerdem kommt vielen Programmierern die Arbeitsweise auf dem definierten Datenmodell entgegen. Dazu bietet StAX eine cursor-basierte und iterator-basierte API an. Aus der Benennung der APIs ist bereits ersichtlich, dass StAX als Mehrschritt-Pull-Parser eingestuft werden muss. Die beiden APIs unterscheiden sich darin, wie Elemente des XML-Dokuments geliefert werden.

Cursor-API Bei der Verwendung der Cursor-API bildet das Interface `XMLStreamReader` die Basis für das Parsen. Mit Hilfe der Methoden des Interfaces wird ein logischer Cursor über einen Strom von XML-Elementen geführt. In jedem, vom Entwickler ausgelösten Schritt, wird der Cursor von einem ereignisauslösenden Element zum nächsten geschoben. Die möglichen Ereignisse ähneln den Ereignissen des SAX-Parsers (u.a. START_DOCUMENT, START_ELEMENT und COMMENT). Mittels Methoden des XMLStreamReaders-Objekts können nähere Informationen über die Ereignisse, wie zum Beispiel die Anzahl der Attribute (`getAttributCount()`) und die Attributwerte

(`getAttributValue()`) selbst sowie Ereignistyp (`getEventType()`) und der Name des aktuellen Elements (`getLocalName()`) festgestellt werden.

Beispiel 5 zeigt wie diese Elemente eingesetzt werden können. Das Programm durchläuft ein XML-Dokument und gibt die Namen der verschiedenen Startelemente aus.

Beispiel 1 Einsatz der Cursor-API [8]

```
URL u = new URL("http://www.cafeconleche.org/");
InputStream in = u.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);

while (true) {
    int event = parser.next();
    if (event == XMLStreamConstants.END_DOCUMENT) {
       parser.close();
       break;
    }
    if (event == XMLStreamConstants.START_ELEMENT) {
        System.out.println(parser.getLocalName());
    }
}
```

Weiterhin unterstreicht das Beispiel, wie wenig Objekte zur Verarbeitung erzeugt werden müssen. Daraus resultiert die große Effizienz dieser Verarbeitungsmethode. Die Verarbeitung auf diese Weise passt jedoch kaum in die objektorientierte Softwareentwicklung [8].

Iterator-API Das zentrale Interface der Iterator-API ist `XMLEventReader`. Es wird verwendet, um über einen Strom von Ereignisobjekten zu iterieren. Die Ereignisobjekte enthalten die nötigen Informationen zu den Ereignissen. Dies ist ein wichtiger Unterschied zur Verarbeitung mit Hilfe der Cursor-API, in der der Parser die Informationen zum Ereignis verwaltet. Diese Variation ermöglicht es, die Ereignisobjekte objektorientiert zu verarbeiten. Die über die Ereignisobjekte zugänglichen Informationen entsprechen den Informationen, die im Zusammenhang mit der Cursor-API erreichbar sind.

XMLEventReader-Objekte erben von `java.util.Iterator`. Mit Hilfe der geerbten Methoden `hasNext()` und `next()` kann auf dem Strom von Ereignissen gearbeitet werden. Darüber hinaus bietet das Interface `XMLEventReader` Methoden, um zum nächsten öffnenden oder schließenden Tag zu springen (`nextTag()`). Die Methode liefert ein Start- oder EndElement. Die Methode `getElementText()` liefert den Inhalt des Knotens zurück. Mit Hilfe der `peek()`-Methode wird das nächste Ereignis im Strom ermittelt, ohne es aus dem Strom zu entfernen.

Beispiel 2 demonstriert wie mit Hilfe der `XMLEventReader`-Methoden die Definition eines Icon-Elements erreicht wird. Nach der Initiierung der `XMLInputFactory` und des `XMLEventReaders` liefert die `peek()`-Methode das nächste Ereignis in Form eines `XMLEvent`-Objekts. Falls dieses Ereignis durch ein öffnendes Tag ausgelöst wurde, wird geprüft, ob es ich um den Beginn eines Icon-Tags handelt. In diesem Fall wird der Inhalt des Icon-Elements durch die Methode `getElementText()` bestimmt. Im Anschluss wird die Iteration beendet. Handelt es sich nicht um ein Startelement für ein Icon, wird das nächste Event (`nextEvent()`) bestimmt [9].

Beispiel 2 Beispiel zum Einsatz von peek() und nextEvent() [9]

```
final QName ICON = new QName("http://www.w3.org/2005/Atom", "icon");
URL url = new URL(uri);
InputStream input = url.openStream();

XMLInputFactory factory = XMLInputFactory.newInstance();
XMLEventReader reader = factory.createXMLEventReader(uri, input);
try {
      while (reader.hasNext()) {
            XMLEvent event = reader.peek();
            if (event.isStartElement()) {
                  StartElement start = event.asStartElement();
                  if (ICON.equals(start.getName())) {
                        System.out.println(reader.getElementText());
                        break;
                  }
            }
            reader.nextEvent();
      }
} finally {
      reader.close();
}
input.close();
```

Filter Mit Hilfe von Filtern ist es möglich, die Menge der vom Parser berückstichtigten Ereignisse einzuschränken. Durch Implementierung der accept()-Methode der Klassen `javax.xml.stream.EventFilter` kann eine beliebige Bedingung formuliert werden. Der in Beispiel 3 präsentierter Filter filtert alle schließenden Tags. Er wird beim Initialisieren des XMLEventReaders als Parameter für das `factory`-Objekt übergeben [10].

Beispiel 3 Beispiel für den Einsatz von Ereignisfiltern [10]

```
package com.javatutor.insel.stax;

import javax.xml.stream.EventFilter;
import javax.xml.stream.events.XMLEvent;

public class PartyEventFilter implements EventFilter
{
  public boolean accept( XMLEvent event )
  {
    return ! event.isEndElement();
  }
}

// Definition eines XMLEventReaders unter Berücksichtigung
// des Filters
XMLEventReader filteredParser = factory.createFilteredReader (
    parser, new PartyEventFilter() );
```

Filter stehen sowohl in der Iterator API wie auch der Cursor API zur Verfügung.

Schreiben StAX ist nicht auf das Lesen von XML-Daten beschränkt. Sowohl in der Cursor- wie auch in der Iterator-API ist es möglich, Daten zu schreiben. In der Cursor API werden die Objekte mit Hilfe der `XMLStreamWriter`-Klasse geschrieben. Während des Schreibvorgangs werden keine zusätzlichen Objekte erzeugt, so dass wiederholtes Schreiben von Daten eine Wiederholung des Programmcodes nötig macht. Beispiel 4 zeigt, wie ein Schreiben eines Dokuments mit Elementen und Attributen umgesetzt wird. Über das `XMLStreamWriter`-Objekt wird mit Hilfe der Methoden writeStartDocument(), writeStartElement() und writeAttribute() die Knotenhierarchie erzeugt. Bis auf das Schreiben der Attribute muss jedes Element durch Aufruf einer writeEnd*-Methode geschlossen werden.

Beispiel 4 Schreiben mit Hilfe der Cursor-API [10]

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(
                            new FileOutputStream( "c:/party.xml" ));
// Der XML-Header wird erzeugt
writer.writeStartDocument();
// Zuerst wird das Wurzelelement mit Attribut geschrieben
  writer.writeStartElement( "party" );
```

```
    writer.writeAttribute( "datum", "31.12.01" );
    // Unter dieses Element wird das Element gast mit
    // einem Attribut erzeugt
    writer.writeStartElement( "gast" );
      writer.writeAttribute( "name", "Albert Angsthase" );
    writer.writeEndElement();
  writer.writeEndElement();
writer.writeEndDocument();
writer.close();
```

Das Schreiben mit Hilfe der Iterator-API wird durch die Klasse `XMLEventWriter` realisiert. Zum Schreiben der Daten müssen XMLEvent-Objekte erzeugt werden. Sie werden im Anschluss über ein XMLEventWriter-Objekt geschrieben. Beispiel 5 illustriert den Umgang mit der `XMLEventWriter`-Klasse. Nach der Initialisierung der Factory- und Writer-Objekte werden die Ereignisobjekte erzeugt. Im letzen Schritt werden die Daten der Objekte mit Hilfe des writer-Objekts geschrieben [10].

Beispiel 5 Schreiben mit Hilfe der Iterator-API [10]

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLEventWriter writer = outputFactory.createXMLEventWriter(
   new FileOutputStream( "c:/party.xml" ));
XMLEventFactory eventFactory = XMLEventFactory.newInstance();

XMLEvent header = eventFactory.createStartDocument();

XMLEvent startRoot = eventFactory.createStartElement
                                ( "", "", "party" );
XMLEvent datumAttribut = eventFactory.createAttribute
                                ( "datum", "31.12.01" );
XMLEvent endRoot = eventFactory.createEndElement
                                ( "", "", "party" );
XMLEvent startGast = eventFactory.createStartElement
                                ( "", "", "gast" );
XMLEvent name = eventFactory.createAttribute
                                ( "name",  "Albert Angsthase" );
XMLEvent endGast = eventFactory.createEndElement
                                ( "", "", "gast" );
XMLEvent endDocument = eventFactory.createEndDocument();

// Schreiben der Struktur
writer.add( header );
writer.add( startRoot );
    writer.add(datumAttribut);
```

```
    writer.add( startGast );
      writer.add( name );
    writer.add( endGast );
  writer.add( endRoot );
writer.add( endDocument );
writer.close();
```

Bewertung StAX ist der neueste Ansatz für XML-Parser und bietet eine dritte Möglichkeit auf das XML Infoset zuzugreifen. Die beiden API-Realisierungen bieten ein breites Einsatzfeld, je nach vorhandenen Ressourcen beziehungsweise flexiblem Zugriff auf erzeugte Objekte. Wie gezeigt, füllt StAX eine größe Lücke in der SAX-Spezifikation und bietet die Möglichkeit des Schreibens von XML-Dokumenten.

2.4 Zusammenfassung

Die drei vorgestellten Parserkonzepte repräsentieren die wichtigesten Möglichkeiten für den Zugriff auf das XML-Infoset. SAX und StAX als Vertreter der ereignisorientierten Parser teilen sich den selben Anwendungsbereich. Sie profitieren von ihrer Ressourceneffizienz. Darüber hinaus bietet StAX die Möglichkeit Daten zu schreiben und komplettiert den Leistungsumfang damit. Durch das Angebot der Cursor- und Iterator API vergrößert sich die Flexibilität des Einsatzes von StAX.

Die DOM API unterscheidet sich grundlegend. Die Verarbeitung von Daten, die im Speicher liegen, ist enorm flexibel. Vor allem ist es möglich wiederholt in beliebiger Reihenfolge auf die Daten zuzugreifen. Diesen Luxus erkauft man sich jedoch durch eine sehr speicherintensiver Abbildung der Daten in ein Datenmodell im Hauptspeicher. In Folge dessen eignet sich der Einsatz der DOM API nur bei kleineren Datenmengen.

3 XML-Anfragesprachen

XML wurde zunächst als reines Datenrepräsentationsformat zwischen 1996 und 1998 entwickelt. Die Entwicklung der XML-Verarbeitung wurde einerseits von der Internetgemeinde und andererseits von der Datenbankgemeinde inspiriert. Erstere entwickelte vor allem Sprachen, die zur Transformation von Dokumenten geeignet sind (z.B.: XSLT). Die Datenbankgemeinde hingegen entwickelte Anfragesprachen ähnlich den aus den objektorientierten und relationalen Datenbanksystemen bekannten Sprachen OQL und SQL.
Die Entwicklung der XML-Anfragesprachen begann bereits 1997 mit Lorel[13]. Unter gegenseitiger Beeinflussung entstanden Alternativvorschläge (XML-QL[14], XQL[15], XML-GL[16]) mit zum Teil grundlegenden Unterschieden [11]. Einerseits gibt es Sprachen, die auf musterbasierten Ansätzen (Query-by-Example) beruhen (XML-QL, 1998), andererseits existieren navigierende Ansätze (XQL, 1998). Aus diesen Sprachen entwickelte sich im Jahre 2000 die Sprache QUILT,

auf der der heutige W3C Standard XQuery basiert. Der zeitliche Verlauf und die Beziehung der Sprachen zueinander ist in Abbildung 5 dargestellt.

In diesem Kapitel sollen je ein Beispiel für eine Sprache mit navigierendem sowie musterbasiertem Ansatz am Beispiel von XQuery/XPath und Xcerpt vorgestellt werden.

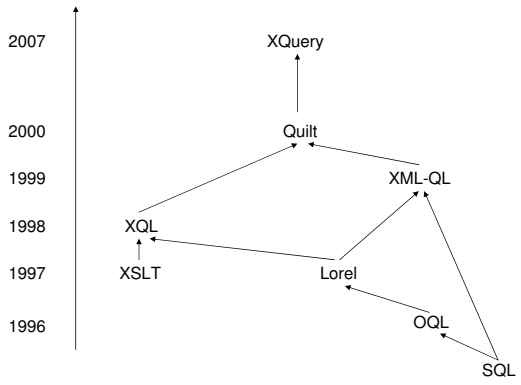

Abbildung 5. Entwicklung der Abfragesprachen [11]

3.1 XPath

Die Sprache XPath wurde 1999 als Standard vom W3C vorgestellt. Sie dient dazu, den gemeinsamen Bedarf der Sprachen XPointer[17] und XSLT[18], Teile von XML-Daten zu adressieren, zu decken. Im weiteren Verlauf mündete die Entwicklung von XPath im Standard XPath 2.0 (2007), der die Sprache vor allem an die Erfordernisse von XQuery anpasst, welches auf XPath aufbaut. Die W3C-Standards verbindet ein gemeinsames Datenmodell (Kap. 1.2), das eine gemeinsame Grundlage für die Verarbeitung von XML-Daten bereitstellt.

Datenmodell Das Datenmodell von XPath ähnelt stark dem hierarchischen Modell des Document Object Models (s. Kap. 1.2). Bis auf die Namensraumknoten existieren alle DOM-Knotentypen in einer Entsprechung auch im XPath-Datenmodell. Seit XPath 2.0 ist die Namensraum-Achse „deprecated“ und wird nur noch aus Kompatibilitätsgründen zu XPath 1.0 unterstützt. In XPath 2.0

sind die Informationen bezüglich der Namensräume ausschließlich durch Funktionen (s. Kap 3.5) abrufbar [12]. Eine weitere Besonderheit der Namensraum- und Attributknoten ist, dass sie nicht als Kindknoten ihrer Element-Elternknoten betrachtet werden. Außerdem ist zum Document Object Model anzufügen, dass der Zeichenkettenwert(string-value) eines Knotens die Konkatenation der Werte der Textknoten der Kindknoten darstellt. Damit liefert die Methode andere Ergebnisse als die Methode `nodeValue()` im DOM, die dort für Element- und Dokumentknoten „null“ liefert [19].

Datentypen Mit XPath 2.0 wird vollständige Kompatibilität zu XML Schema eingeführt. Dadurch vervielfachen sich die zur Verfügung stehenden Datentypen von bisher vier (*node-set*, *booelan*, *number* und *string*) auf 19 primitive Datentypen. Darüber hinaus wurde der Umfang der angebotenen Funktionen und Operatoren erweitert (s. Kap. 3.5).

Sequenzen Wie bereits in der Einführung dieses Kapitels angedeutet, dient XPath zur Adressierung von Teilen eines XML-Dokuments. Durch die mit der Einführung von XPath 2.0 zur Verfügung stehenden Ausdrücke, Funktionen, Operatoren und erweiterten Datentypen hat die Sprache an Mächtigkeit gewonnen und erlaubt weitergehende Operationen als nur die Adressierung von Teilen eines XML-Dokuments. Seit XPath 2.0 werden Ergebnismengen von Knoten nicht mehr als Node-Sets, also als ungeordnete Knotenmengen, sondern als Sequenzen bezeichnet. In XPath wird nun alles als Sequenz betrachtet und jeder Pfadausdrucksschritt liefert eine Sequenz zurück. Eine Sequenz ist eine geordnete Menge, die keine Untermengen besitzen kann. So sind die Sequenzen (1, 2,(3, 4)) und (1, 2, 3, 4) identisch. Weiterhin sind, im Gegensatz zu Node-Sets, Duplikate möglich. Die Einführung von Sequenzen wurde unter anderem durch Neuerungen in XSLT 2.0 nötig. Seit XSLT 2.0 ist die Verarbeitungsreihenfolge nicht mehr auf die Dokumentordnung beschränkt [20]. Für die Umsetzung anderer Reihenfolge wurde es daher erforderlich sortierte Mengen einzuführen.

Aus Kompatibilitätsgründen ist es notwendig, dass die Möglichkeit besteht Node-Sets durch Sequenzen zu simulieren. Node-Sets sind unsortiert, werden jedoch standardmäßig in „document order“ verarbeitet. Um Abwärtskompatibilität zu XPath 1.0 zu garantieren, sind die Elemente zurückgegebender Sequenzen somit in Dokumentordnung (document order) sortiert [20].

Pfadausdrücke Pfadausdrücke stellen das wichtigste Konstrukt in XPath dar. Sie dienen der Adressierung von Knotensequenzen. Die Adressierung orientiert sich am hierarchischen Aufbau der Repräsentation des XML-Dokuments und lässt sich in absolute und relative Pfade unterteilen. Erstere adressieren ausgehend vom Wurzelknoten des Dokuments. Absolute Pfade werden durch ein `"/"` (Slash) eingeleitet. Relative Pfade adressieren ausgehend vom aktuellen Kontextknoten aus. Ein Pfadausdruck besteht aus mehreren Schritten. Jeder Schritt wiederum aus den drei Bestandteilen *Achse*, *Knotentest* und *Prädikat*. Als erstes

wird die Achse ausgewählt. Darauf folgen beliebig viele Knotentests und Prädikate.

```
(1)Pfadausdruck     ::=  ("/" RelativerPfadausd?)|
                         ("//" RelativerPfadausd)|RelativerPfadausd
(2)RelativerPfadausd::=   Schritt (("/" | "//") Schritt)*
(3)Schritt          ::=   AchsenTest PrädikatListe
(4)AchsenTest       ::=     Achse (KnotenTest*)
```

Abbildung 6. Aufbau eines Pfadausdrucks

Die Navigation in der Repräsentation des XML-Dokuments orientiert sich an gegebenen Achsen und selektiert Schritt für Schritt Knotensequenzen, die ausgewählt werden sollen. Die einzelnen Schritte werden durch Schrägstrich (Slash "/") voneinander getrennt (vgl. Abb. 6). Jede Achsenbezeichnung ("Achse") besteht aus einem Achsennamen, dem ein doppelter Doppelpunkt ("::") folgt. Die doppelten Slashes "//" sind eine Abkürzung (s. unten). Es werden vorwärts- und rückwärtsorientierte Achsen unterschieden. Die zur Verfügung stehenden Achsen sind:

Die *self*-Achse wählt den Kontextknoten selbst aus. Die *child*-Achse enthält alle direkten Kinder des Kontextknotens. Darunter können sich neben Elementknoten auch Knoten anderen Typs befinden (Textknoten, Kommentarknoten oder Processing Instruction Knoten), die über Knotentests identifiziert werden können (s. unten), jedoch keine Attributknoten. Die *parent*-Achse wählt den Vater des Kontextknotens aus. Ist der Vater der Dokumentwurzelknoten, so ist die Ergebnissequenz leer. Die *ancestor*-Achse umfasst alle Vorfahren. Dazu zählen der Elternknoten, dessen Elternknoten sowie alle Elternknoten bis zum Dokumentwurzelknoten. Die *ancestor-or-self*-Achse erklärt sich als Vereinigung der Elementknoten, die von der ancestor-Achse und der self-Achse ausgewählt werden. Sie wählt also alle Elternknoten bis zum Wurzelelement sowie den Kontextknoten aus. Im Gegensatz zur ancestor-Achse enthält sie immer auch den Wurzelknoten, der bei der ancestor-Achse entfällt, wenn der Kontextknoten selbst der Wurzelknoten ist. Die *descendant*-Achse entspricht der transitiven Hülle der child-Achse. Wie für die child-Achse gilt auch hier, dass Knoten verschiedenen Typs ausgewählt werden können. Die *descendant-or-self*-Achse erweitert die descendant-Achse um den Kontextknoten. Die *following*-Achse umfasst alle nachfolgenden Knoten in Dokumentordnung. Damit ist das Ergebnis abhängig von der Ordnung der Elemente im Dokument. Durch die *following-sibling*-Achse werden alle Knoten erfasst, die den gleichen Elternknoten wie der Kontextknoten haben und in Dokumentordnung nach dem Kontextknoten folgen. Die *preceding*-Achse ist das Gegenstück der following-Achse. Sie umfasst alle Knoten, die in Dokumentordnung vor dem Kontextknoten liegen. Die *preceding-sibling*-Achse ist das Gegenstück zur following-sibling-Achse und umfasst somit alle Knoten,

die in Dokumentordnung vor dem Kontextknoten liegen und den gleichen Vaterknoten besitzen. Über die *attribute*-Achse werden alle Attribute des Kontextknotens ausgewählt. Besitzt der Knoten keine Attribute, ist die Achse leer. Es ist zu beachten, dass Attributknoten keine Ordnung haben. Syntaktisch gleichen Attributknoten den Namensraumknoten. Letztere sind jedoch nicht über die Attribut-Achse ansprechbar. Die *namespace*-Achse selektiert alle Namensräume, die für den Kontextknoten gelten. Dazu zählen auch die Namensräume, die durch Vererbung hinzukommen. Namensraumknoten haben keine Ordnung zueinander [21].

Wie in der Einleitung zu diesem Abschnitt angedeutet, lassen sich die Achsen in vorwärts- und rückwärtsorientierte Achse einteilen. Nur die Attribut- und Namensraumachsen besitzen keine Orientierung. Die Übersicht in Tabelle 1 gibt dazu einen Überblick. Der primäre Knotentyp (*principle node kind*) bezeichnet die Art der standardmäßig zu erreichenden Knoten (vgl. Knotentests, s.unten).

Axenbezeichnung	Richtung	primärer Knotentyp
ancestor	rückwärts	element
ancestor-or-self	rückwärts	element
attribut	-	attribut
child	vorwärts	element
descendant	vorwärts	element
descendant-or-self	vorwärts	element
following	vorwärts	element
following-sibling	vorwärts	element
namespace	-	namespace
parent	vorwärts	element
preceding	rückwärts	element
preceding-sibling	rückwärts	element
self	vorwärts	element

Tabelle 1. Übersicht über Achsen, ihre Orientierung und den primären Knotentyp

Für die Formulierung von häufig verwendeten Achsen, gibt es in XPath Abkürzungen (vgl. Tab. 2). Die am häufigsten genutzte Achse ist die child-Achse. Wird keine Achse explizit angegeben, wird sie ausgewählt. Für die attribut-Achse gibt es die Kurzform `@`. Die Auswahl des Kontextknotens kann statt mit `self::node()` mit Hilfe von `.` abgekürzt werden. Um den Vaterknoten zu erreichen, genügt die Abkürzung `..`. Die Kurzform `//` entspricht dem Ausdruck `/descendant-or-self::node()/`.

Knotentests Nach der Auswahl der Achsen folgt die Auswertung der Knotentests. Über Knotentests lässt sich die Menge ausgewählter Knoten reduzieren. Dabei werden Namensprüfungen (name tests) und Typtest (kind tests) unter-

child::	leer
attribut::	@
self::node()	.
parent::node()	..
/descendant-or-self::node()/	//

Tabelle 2. Kurzformen für häufig genutzte Achsen

schieden. Namensprüfungen sind erfolgreich, wenn der Knotenname mit dem gesuchten Namen übereinstimmt. Typentests hingegen wählen alle Knoten eines bestimmten Knotentyps aus. Die möglichen Typentests sind in Tabelle 3 beschrieben. Dabei ist zu beachten, dass die Tests document-node(), attribut() und element() durch Parameter verfeinert werden können. Beispielsweise erfüllen den Test `element(person)` nur jene Elementknoten, die den Namen "Person" tragen. Des Weiteren ist zu beachten, dass jeder Achse eines Knotentyps (*principle node kind*) zugeordnet ist (vgl. Tab. 1). Entsprechend dieses Knotentyps werden Knotensequenzen ausgewählt, wenn kein spezifischer Knotentest das Ergebnis anders definiert.

Test	erfüllt für
*	jeden Knoten des primären Knotentyps
node()	jeden Knoten
text()	jeden Textknoten
comment()	jeden Kommentarknoten
processing-instruction()	jeden Processing-Instructions Knoten
attribute()	jeden Attributknoten
element()	jeden Elementknoten
document-node()	jeden Dokumentknoten

Tabelle 3. Übersicht über Knotentests

Prädikate Der dritte Teil der Auswertung eines Pfadausdruckschritts besteht aus der Auswertung beliebig vieler Prädikate. Ein Prädikat kann ein beliebiger XPath-Ausdruck sein. Alle Knoten für die ein Prädikat wahr ist, werden ausgewählt. Dabei gilt, dass ein Ausdruck dessen Ergebnis ein atomarer numerischer Wert ist oder von einem numerischen Wert abgeleitet ist wahr ist, wenn dieser Wert gleich der Kontextposition ist. Die Kontextposition entspricht der Position des Kontextknotens in der Eingabesequenz. Ist das Ergebnis keine Zahl, so wird das Ergebnis, wie bei einem Aufruf der Funktion `boolean`, bestimmt (s. unten).

Es folgen einige Beispiele, die den Einsatz der verkürzten Syntax sowie den Einsatz von Prädikaten verdeutlichen:

ursprüngliche Version	kurze Version
attribute::name	@name
child::para[position()=1]	para[1]
/child::doc/child::chap[position()=5]/child::sect[position()=2]	/doc/chap[5]/sect[2]
child::para[attribute::type='warning'][position()=5]	para[@type='warning'][5]

Tabelle 4. Beispiele zur verkürzten Syntax [22]

Wie man sieht, führen die Abkürzungen zu einer besseren Lesbarkeit und zu kürzeren Ausdrücken [21].

Allgemeine Ausdrücke Der Ausdruck ist das allgemeinste Konstrukt in XPath. Die bereits vorgestellten Pfadausdrücke stellen nur eine besondere Form des Ausdrucks dar. XPath-Ausdrücke bestehen aus Operatoren und Operanden, die wiederum Ausdrücke sein können. In diesem Abschnitt sollen zwei interessante Formen von Ausdrücken vorgestellt werden, die mit XPath 2.0 eingeführt worden sind - den Konditionalausdruck und den quantifizierenden Ausdruck.

Konditionalausdruck Ähnlich vieler bekannter Programmiersprachen unterstützt XPath seit der Version 2.0 ein if-Konstrukt.

`IfExpr ::= " if " "(" Expr ")" " then " ExprSingle " else " ExprSingle`
Falls `Expr` gilt, wird der erste Ausdruck ausgewählt, andernfalls der Zweite.

Beispiel 1: Konditionalausdruck
```
if ($part/@discounted)
then $part/wholesale
else $part/retail
```

Hat ein part-Element ein Attribut `discounted`, wird der normale Preis (`wholesale`) ausgewählt. Gibt es keinen Rabatt, wird der Wert von `retail` ermittelt [22].

Quantifizierender Ausdruck (quantified expression) XPath verfügt über zwei Quantoren, den Existenzquantor und den Allquantor.

`QuantifiedExpr ::= (" some " | " every ") "$ " VarName " in " ExprSingle ( ", " "$ "VarName "in " ExprSingle)* "satisfies " ExprSingle`

Der Ausdruck beginnt mit einem der Schlüsselworte `some` beziehungsweise `every` für existenzielle- bzw. universelle Quantifizierung, auf den eine Liste von Ausdrücken zur Bindung an Variablen folgen. Abschließend wird mit dem Schlüsselwort `satisfies` der zu erfüllende Ausdruck eingeleitet.

Ein quantifizierender Ausdruck evaluiert zu wahr oder falsch. Dabei gilt, dass ein Ausdruck mit einem existentiellen Quantifizierer wahr ist, wenn mindestens eine der gebundenen Variablen den Ausdruck erfüllen. Handelt es sich um einen Ausdruck mit einem Allquantor müssen alle Tests erfolgreich verlaufen.

Beispiel 2:
`every $part in /parts/part satisfies $part/@discounted`
Dieser Ausdruck ist wahr, falls jedes part-Element unter einem parts-Element ein Attribut `discounted` besitzt [22].

Funktionen und Operatoren Eine weitere Eigenschaft, die die Sprache erweitert, sind Funktionen. XPath bietet eine Menge an Grundfunktionen, die bei Bedarf um weitere Funktionen erweitert werden kann. Zu den Kernfunktionen, die von jeder XPath-Implementierung unterstützt werden müssen, gehören Funktionen auf Zeichenketten, Zahlen, Zeiten und Wahrheitswerten. Die `boolean`-Funktion ergibt für leere Sequenzen `falsch`. Wenn das erste Element der Sequenz ein Knotenelement ist, wird *wahr* geliefert. Handelt es sich um einen boolschen Wert, wird dieser zurückgegeben. Operanden vom Typ string, anyURI, untypedAtomic sowie einem aus ihnen abgeleiteten Typs lieferen *falsch*, falls der Operand null Zeichen lang ist. Handelt es sich um einen numerischen Wert, ist das Ergebnis falsch, falls der Operand gleich 0 ist oder es sich nicht um eine Zahl handelt, andernfalls *wahr*. In allen anderen Fällen liefert die Funktion *falsch*. Hinzukommen eine Menge von Operatoren. Eine Übersicht über alle Funktionen und Operatoren bietet [22, 25].

3.2 XQuery

Im Dezember 1998 organisierte das W3C einen Workshop zu XML-Anfragesprachen. Als Resultat der Konferenz wurde eine Arbeitsgruppe gegründet, die ein Datenmodell sowie eine zugehörige XML-Anfragesprache spezifizieren sollte. Als Randbedingung für die Entwicklung galt, dass die Ergebnisse kompatibel zu vorhandenen W3C-Standards sein müssen. Auf Grund der Beliebtheit von XPath war es wünschenswert, kompatibel zu XPath 1.0 zu bleiben. Tatsächlich ist XPath zu einer Untermenge von XQuery geworden. XPath basierte noch nicht auf XML Schema, das seit der ersten XPath-Version eine zunehmend größere Bedeutung erlangt hat. In Folge dessen wurden im Laufe der Entwicklung von XQuery Erweiterungen definiert, die XPath an die neuen Anforderungen anpassen. Auf Grund der vielseitigen Verwendung von XPath in anderen Standards bestand ein erheblicher Teil der Arbeit darin, die Abhängigkeiten und Anforderungen anderer W3C-Standards zu berücksichtigen. Standards wie XSLT wurden angepasst und erschienen zusammen mit XQuery Anfang 2007 in neuen

Versionen.

Neben den Abhängigkeiten zu bestehenden W3C-Standards wurde die Entwicklung von XQuery durch funktionale Anforderungen beeinflusst. Als eine vielseitige Sprache sollte XQuery in der Lage sein, sowhl XML-Dokumente zu verarbeiten, zu denen Schemainformationen in Form von DTDs oder XML Schema vorliegen, als auch Dokumente zu erschließen, für die keine derartigen Metadaten verfügbar sind.

Die Sprache In XPath fehlt die Möglichkeit neue XML-Elemente zu erzeugen. Dieser Mangel wird durch XQuery behoben. XQuery ist eine funktionale Sprache, die aus verschiedenen Ausdrücken besteht. Die Ausdrücke liefern nur Werte zurück, sind also frei von Seiteneffekten. Die einfachste Form des Ausdrucks ist das Literal, das einen atomaren Wert enthält (z.B. 47 als ein Literal vom Typ Integer). Variablen werden durch Dollarzeichen ($) eingeleitet und können in einem LET-Ausdruck gebunden werden, der ein Teil der FLWR-Ausdrücke (FOR, LET, WHERE, RETURN) ist. Ein gültiger FLWR-Ausdruck muss mindestens einen FOR- oder LET-Ausdruck enthalten. Der FOR-Ausdruck bindet jedes Element einer Sequenz einzeln an eine entsprechende Variable. In Beispiel 6 bedeutet das, dass für jedes `custno`-Element die Variable $c einmal belegt wird und damit der return-Ausdruck ausgeführt wird. Der LET-Ausdruck bindet die gesamte Sequenz der vier custno-Elemente an die Variable $c. Daher wird in Beispiel 7 der Konstruktor nur einmal ausgeführt und die custno-Elemente zusammen in ein `customers`-Element verschachtelt.

Beispiel 6 Funktion des FOR-Ausdrucks [24]

```
for $c in document("data/customers.xml")//customer/custno
   return
      <customers>
      {$c}
      </customers>

Liefert
   <customers>
       <custno>9000</custno>
   </customers>
   <customers>
       <custno>1001</custno>
   </customers>
   <customers>
       <custno>1003</custno>
   </customers>
   <customers>
       <custno>2005</custno>
   </customers>
```

Beispiel 7 Funktion des LET-Ausdrucks [24]

```
let $c := document("data/customers.xml")//customer/custno
   return
      <customers>
      {$c}
      </customers>

Liefert
<customers>
       <custno>9000</custno>
       <custno>1001</custno>
       <custno>1003</custno>
       <custno>2005</custno>
</customers>
```

Nach der Definition der Variablen wird der WHERE-Ausdruck ausgewertet. Dieser filtert, ähnlich des WHERE-Konstrukts in SQL das Ergebnis bezüglich beliebiger Bedingungen. Im RETURN-Ausdruck werden die ausgewählten Sequenzen strukturiert und zurückgegeben. Dieser Teil wird Konstruktor genannt. Im einfachsten Fall besteht dieser Abschnitt ausschließlich aus Konstanten und gleicht reinem XML.

Natürlich können auch Variablen im Elementkonstruktor verwendet werden. Variablen, die ersetzt werden sollen beziehungsweise Ausdrücke, die vor der Ausgabe ausgewertet werden müssen, werden in geschweifte Klammern eingeschlossen. In Beispiel 8 werden die Variabeln $s, $i ersetzt und der Ausdruck zur Bestimmung des Maximums ausgewertet.

Beispiel 8 Variablen im Element-Konstruktor [23]

```
<highbid status = '{$s}'>
<itemno>{$i} </itemno>
  <bid-amount>
     {max($bids[itemno = $i]/bid-amount)}
  </bid-amount>
</highbid>
```

Auf jeden FLWR-Ausdruck kann ein `order by`-Ausdruck folgen. Er dient der Sortierung der Ergebnisse Sequenz. `order by` ist nicht Teil eines FLWR-Ausdrucks, sondern ein weiterer XQuery Ausdruck mit dessen Hilfe jede beliebige Sequenz sortiert werden kann [23].

Beispiel 9 Sortier-Operator [24]

```
for $e in $emps
return
  <emp>
```

```
  {
   $e/name,
    <pay> {$e/salary + $e/commission + $e/bonus} </pay>
  }
  </emp>
order by (pay)
```

Das Beispiel 9 zeigt die Anwendung des *order by*-Ausdrucks. Aus der gegebenen `emps`-Sequenz werden das Gehalt, Komissionen und Boni entnohmen und für jeden Mitarbeiter (emp) addiert. Die Mitarbeiter werden anhand dieser Summe sortiert.

Wie bereits mehrfach angedeutet, verwendet XQuery XPath zur Adressierung von Knoten. Dadurch stehen die aus XPath bekannten Prädikate, Funktionen, Quantoren und Konditionalausdrücke auch hier zur Verfügung. Im Gegensatz zu vielen der verbreiteten Programmiersprachen ist es in XQuery jedoch nicht möglich, Funktionen zu überladen. Je nach Kontext konvertiert XQuery ungetypte Parameterwerte in das benötigte Format der jeweiligen Funktion. Folglich ist es nicht notwendig, eine Funktion für Parameter vom Typ `decimal` als auch `character` o.ä. anzulegen. Da das System über den Datentyp entscheidet, ist es hinderlich, wenn eine Funktion mit verschiedenen Parametertypen existieren würde. Aus Kompatibilitätsgründen zu XPath 1.0 gibt es in der Kernbibliothek von XQuery Funktionen die überladen sind. Es ist jedoch nicht möglich dies mit benutzerdefinierte Funktionen zu tun.

Implementierung von Joins Mit Hilfe der FLWR-Ausdrücke ist es möglich, die aus relationalen Datenbanken bekannten Joins zu implementieren. Ein Join arbeitet üblicherweise auf Daten mehrerer Tabellen. Im XML-Umfeld bedeutet dies Daten eines oder mehrerer Dokumente zu verarbeiten und zu verknüpfen.

Das Beispiel 10 zeigt zwei verschachtelte FOR-Ausdrücke. Die Variablen $j und $p enthalten zwei Sequenzen von Elementen. Der folgende WHERE-Ausdruck reduziert das Ergebnis auf jene `item`-Objekte, die die gleichen Werte für `itemno` haben. Diese Art des Joins wird in relationalen Datenbanken als *inner join* bezeichnet. Weiterhin ist zu beachten, dass die Variable $p zweimal gebunden wird. Die zweite Bindung der Variablen $p ist so lange gültig, wie der Gültigkeitsbereich der Variablen (scope) nicht verlassen wird.

Beispiel 10 Beispiel zum inner join [24]

```
for $p in document("data/PO.xml")//po
   return
      <new_orders>
      {
      for $j in $p//item
      for $p in document("data/items.xml")//item
      where $p/itemno=$j/itemno
```

```
return
   <item>{$p/description/text()}</item>
}
</new_orders>
```

Beispiel 11 beschreibt die Umsetzung eines *left outer joins*. Dabei werden alle Informationen mindestens eines der Quelldokumente erhalten, auch wenn es keine passenden Elemente zur Verknüpfung gibt. In der Anfrage werden zwei FOR-Ausdrücke verschachtelt. Es werden alle Kunden (customer) mit ihren Bestellungen aufgeführt. Sind einem Kunden keine Bestellungen zugeordnet, bleibt der innere FLWR-Ausdruck ergebnislos und es wird mit der Iteration über die Sequenz der Elemente, die über $u erreichbar sind, fortgefahren [24].

Beispiel 11 Left Outer Join [24]

```
for $u in document("data/customers.xml")//customer
   return
      <customer id=$u/custno>
         <name>{$u//firstname/text()}   {$u//lastname/text()}</name>
         {
           for $p in document("data/PO.xml")//po
           where $u/custno = $p//custno
           return
           <po>{$p/@id}</po>
         }
      </customer>
```

Was XQuery nicht ist und nicht kann Durch den großen Erfolg von XQuery ist der Eindruck entstanden, dass XQuery andere Technologien ersetzen wird. XSLT und XQuery sind beide turing-vollständig [26], besitzen somit die Fähigkeit, alles durch herkömmliche Programmiersprachen berechenbare zu berechnen. Dennoch soll XQuery XSLT nicht ersetzen. Beide Standards wurden parallel entwickelt und haben ihre Stärken in verschiedenen Bereichen. XSLT eignet sich vor allem im dokumentorientierten Einsatz von XML, wohingegen XQuery im datenorientierten Bereich vorzuziehen ist. Insbesondere für die reine Selektion von Daten, die mit Hilfe von XPath-Ausdrücken erreicht werden, ist XQuery als Obermenge von XPath von Vorteil. Auch im Bereich von XML-Datenbanken ist XQuery die meist passendere Wahl [27].

Ein Vorurteil besagt, dass XQuery nicht performant genug sei, um große Datenmengen zu verarbeiten. Skalierbarkeit und Performanz hängen stark vom Einsatzzweck und der XQuery-Implementierung ab. Durch die Flexibilität von XQuery gibt es sehr unterschiedliche Einsatzmöglichkeiten, die verschiedene Optimierungen ermöglichen. Produkte, die auf das Veröffentlichen von Webinhalten spezialisiert sind, sind auf kleine Dokumente optimiert. Implementierungen im Umfeld von XML-Datenbanken wiederum sind speziell auf große Datenmengen

ausgelegt. Ein generelles Urteil über Performanz und Skalierbarkeit von XQuery ist somit nicht möglich.

Die XQuery-Spezifikation umfasst keinen Update Mechanismus. Allerdings gibt es verschiedene Ansätze, die Aktualisierungen von XML-Daten ermöglichen. XUpdate ist ein XML-Update-Sprache, die im Rahmen des XML:DB-Projektes[29] entwickelt wurde. Die letzten Entwicklungen gehen allerdings bereits zurück bis ins Jahr 2000. Ein aktuellerer Ansatz stellt das XQuery Update Facility[30] dar. Hierbei handelt es sich um einen W3C-Entwurf, der im August 2007 veröffentlicht wurde [28]. Auf die Details dieser Update-Sprachen kann im Rahmen dieser Arbeit nicht eingegangen werden.

3.3 Musterbasierte Anfragesprachen am Beispiel von Xcerpt

Neben den bereits vorgestellten Sprachen, die als etablierte Standards zur Verarbeitung von XML-Daten betrachtet werden können, werden weiterhin Sprachen entwickelt, die zum Teil deutlich andere Ansätze verfolgen.

Die gängige Adressierung anhand von Achsen und Nachbarschaftsbeziehung (s. Kap. 2.3) hat ihre Stärke in der Formulierung von kurzen Anfragen. Komplexe Suchanfragen werden im Vergleich schnell unübersichtlich. Der musterbasierte Ansatz beschreibt an einem Beispiel (query-by-example), welche Daten ausgewählt werden sollen.

Beispiel 12 Anfrage in Pseudosyntax

```
<address-book>
   <person>
      <name>
         <first>Mickey</first>
         <last>Mouse</last>
      </name>
      <phone>$PHONE</phone>
      <email>$EMAIL</email>
   </person>
</address-book>
```

Im Beispiel 12 werden die Telefonnummer und Mailadresse von Mickey Mouse gesucht und an die Variablen $PHONE und $EMAIL gebunden.

Der musterbasierte Ansatz bietet die Vorteile, dass Anfragen dieser Art ähnlich strukturiert sind wie die Daten selbst und somit leicht zu verstehen sind. Außerdem soll die größere Abstraktion größeres Potential für automatische Optimierungen bieten [32].

Als Beispiel für solche Sprache soll hier die Sprache Xcerpt [32] vorgestellt werden. Sie wurde 2004 von einer Forschungsgruppe der LMU München vorgestellt. Inzwischen wird die Sprache im Rahmen des EU Network of Excellence

REWERSE[4] weiterentwickelt. Im Gegensatz zu den bisher präsentierten Sprachen verfolgt Xcerpt einen regel- und musterbasierten Ansatz. Darüber hinaus ist Xcerpt nicht an XML als Datenmodell gebunden, sondern in der Lage, jede Art von semistrukturierten Daten zu verarbeiten. Durch diese Fähigkeit wird die Sprache äußerst interessant zur Lösung verschiedener Anforderungen aus dem Bereich des Semantic Web. In diesem Themenfeld müssen XML-Daten sowie RDF und andere Repräsentationsformen von Metadaten verarbeitet werden können. XQuery und XSLT bieten hierzu Erweiterungen an. Dennoch bleiben sie stark spezialisiert und bieten keine einheitliche Benutzung für die verschiedenen Ansätze an. Xcerpt hingegen möchte die Verarbeitung aller Formate in seinem Sprachumfang verbinden, eine Eigenschaft die als *format versatility* bezeichnet wird. Als weitere Aspekte wurden in neueren Arbeiten *schema versatility* und *representational versatility* betont. Unter schema versatility versteht man die Fähigkeit, Daten zu verarbeiten, die in unterschiedlichen Schemata vorliegen. In semistrukturierten Daten ist es außerdem möglich Daten semantisch gleichen Inhalts in unterschiedlicher Weise darzustellen. Beispielsweise können Abschnitte eines Buchs als `sect1` oder auch `section` innerhalb eines XML-Dokuments bezeichnet werden. Je nach Anwendung kann es notwendig sein, beide Elemente als gleichwertig zu betrachten. In einer Sprache für das Web, die representational versatility unterstützt, muss es möglich sein, derartige Zusammenhänge formulieren zu können [31].

Die Entwicklung der Sprache ist geprägt von einer Reihe von Eigenschaften und Anforderungen. Unter referenzieller Transparenz versteht man, dass jeder Ausdruck an jeder Stelle die gleiche Bedeutung hat. XQuery erfüllt diese Eigenschaft nicht, da Ausdrücke relativ zum Kontextknoten betrachtet werden. Antwortabgeschlossenheit fordert, dass jedes Ergebnis auch als Anfrage oder Ergebnis einer Unteranfrage verwendet werden kann.

Ein Xcerpt-Programm besteht aus beliebig vielen Regeln, die der Strukturierung des Anfrageprogramms dienen können. Jede Regel besteht aus einem Anfrageterm und einem Konstruktionsterm. Somit besteht eine klare Trennung zwischen Anfrage und Antwortproduktion. Sie soll die Arbeit mit unterschiedlichen Schemata in Quell- und Zieldokument sowie die Lesbarkeit erleichtern.

Die Regelverkettung (Rule Chaining) erweitert die Sprache um die Möglichkeit Ergebnisdaten einer Regel als Eingabedaten einer weiteren Regel zu verwenden. Auch das erhöht die Lesbarkeit und bildet die Grundlage, Elemente wiederzuwenden [32].

Die Sprache Xcerpt Die Sprache gliedert sich in Datenterme, Anfrageterme und Konstruktionsterme, die in dieser Reihenfolge vorgestellt werden. Diese Unterteilung macht bereits deutlich, dass Anfrage und Antwortkonstruktion streng

[4] http://rewerse.net

getrennt werden. Zum Abschluss wird die Regelverkettung als wichtige Eigenschaft vertieft.

Syntaktische Grundlage
Xcerpt bietet die Möglichkeit geordnete und ungeordnete Terme zu berücksichtigen. Eckige Klammern ([]) stehen für geordnete Terme und geschweifte Klammern ({}) für ungeordnete Terme.

Datenterme Datenterme stellen die Daten eines XML-Dokuments dar. Sie stellen die Datenbasis für die Anfragen dar. Datenterme werden anhand der in Abbildung 7 dargestellten Syntax formuliert. Sie dürfen keine unvollständigen Terme ({{o. [|) umfassen. Abbildung 7 beschreibt die Syntax genauer.

```
1 data-term := ( oid "@" )? <ns-label> <list> .
2 ns-label := (<ns-prefix> ":")? label
3 ns-prefix := label | '"' iri '"' .
4 list := ordered-list | unordered-list .
5 ordered-list := "[" attributes? data-subterms? "]" .
6 unordered-list := "{" attributes? data-subterms? "}" .
7 data-subterms := data-subterm ( "," data-subterm )*
8 data-subterm := data-term | '"' string '"' | number | "^" oid .
9 attributes := "attributes" "" attribute ( "," attribute )* "" .
10 <attribute := ns-label " {" '"' string '"' " }" .
```

Abbildung 7. Syntax eines Ausdrucks

Ein Datenterm besteht nach Abbildung 7 aus einem möglichen Objektidentifizierer (OID), optionalen Namensraumdaten und einem Bezeichner gefolgt von einer geordneten oder ungeordneten Liste von Attributen und Subtermen. Sie sind entweder wieder vollständige Datenterme, OID-Referenzen oder Daten in Form von Zeichenkette oder Zahlen.

Beispiel 13 Ein Ausdruck am Beispiel eines Adressbuchs

```
address-book {
   &o1 @ person {
      name [
         first [ "Mickey" ],
         last [ "Mouse" ]
      ],
      phone [ "19281118" ],
      knows [ ^&o2 ]},
   &o2 @ person {
```

```
      name [
         first [ "Donald" ],
         last [ "Duck" ]
      ],
      knows [ ^&o1 ]
   }
}
```

Beispiel 13 illustiert die Darstellung eines Adressbuchs in der in Abbildung 7 angegebenen Syntax. Das Adressbuch enthält zwei Personen, Mickey Mouse und Donald Duck. Jede Person hat wiederum einen Namen, der sich in Vor- und Nachname unterteilt. Desweiteren hat eine Person eine Telefonnummer. Das Element `knows` dient dazu Bekanntschaften darzustellen. Zu diesem Zweck werden Referenzen auf OIDs verwendet.

Im Gegensatz zu XML bietet Xcerpt kein Konstrukt zur Beschreibung von Attributen an. Stattdessen werden Attribute als Subterme repräsentiert und in einem Subterm `attributes` zusammengefasst (vgl. Bsp. 14) [32].

Beispiel 14 Attributdarstellung in Xcerpt

```
<book year="1995">
 <title>Vikinga Blot</title>
<\book>

book [
 attributes { year { "1995" } },
 title [ "Vikinga Blot" ],
]
```

Aus jedem derart definierten Ausdruck läßt sich ein Graph ableiten. Dabei ist zu beachten, dass jeder Ausdruck aus Unterausdrücken bestehen kann. Der durch Definition 1 definierte Graph besitzt einen Knoten je Ausdruck inklusive der Unterausdrücke. Jeder Ausdruck ist über eine Kante mit seinen Unterausdrücken verbunden. Darüberhinaus wird für jede Referenzbeziehung (`^oid`) eine Kante zum dem Knoten mit dem Identifizierer `&oid` eingeführt.

Definition 1 *Sei e ein semistrukturierter Ausdruck. Der Graph zum Ausdruck e ist definiert als* $G_e = (V, E, r)$ *mit [32]:*

1. *einer Menge V von Knoten als Menge aller Unterausdrücke von e (inklusive e selbst)*
2. *einer Menge* $E \subseteq V \times V \times \mathbb{N}$ *mit:*
   - für alle Ausdrücke $e1, e2, e3 \in V$ *gilt: Ist e2 i-ter Unterausdruck von e1 und eine Referenz der Form* `^oid`*, Ist der Ausdruck e3 der Form oid @ e' mit oid als ein Objektidentifikator (OID) und e' einem Ausdruck, dann ist* $(e1, e3, i) \in E$

- *für alle Ausdrücke* $e1, e2 \in V$*: gilt e2 ist i-ter Unterausdruck von e1 und nicht der Form* `^oid`*, dann gilt* $(e1, e2, i) \in E$*.*

3. *es gibt einen eindeutige Wurzel* $r \in V$ *mit* $r = e$*. Das Label der Knoten ist das Label des Unterausdrucks.*

Der durch den Ausdruck (Bsp. 13) definierte Graph ist in Abbildung 8 dargestellt. Unterausdrücke sind durch [...] abgekürzt.

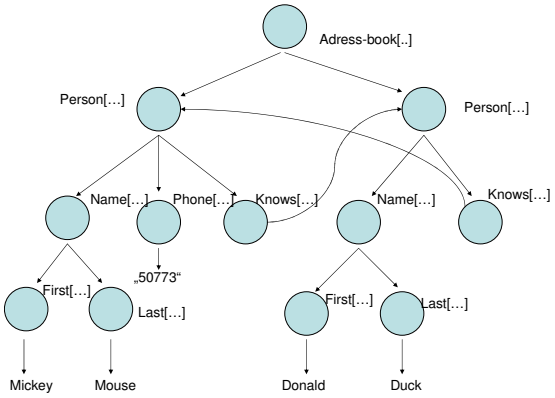

Abbildung 8. Graph zum Adressbuch

Anfrageterme Anfragen dienen der Auswahl von Daten. Anfrageterme beschreiben dazu Muster, die mit Datentermen in Beziehung gesetzt (vgl. Graph Simulation). Im Vergleich zu Datentermen können Anfragen unvollständig sein. Außerdem ist es möglich Subterme in beliebiger Tiefe auszuwählen und Ergebnisse an Variablen zu binden.

Letzteres erlaubt die spätere Verwendung der Ergebnisse im Konstruktionsterm. In Xcerpt werden vier Arten von Variablen unterschieden. *Variablen ohne Einschränkungen* können an jeden Subterm gebunden werden. Diese Variablen werden durch das Schlüsselwort `var` gefolgt von einem Bezeichner eingeleitet.
Variablen mit Einschränkungen beginnen ebenfalls mit `var <bezeichner>`. Die Einschränkung wird daraufhin durch einen Anfrageterm definiert. Damit wird erreicht, dass dieser Variablen nur noch Subterme entsprechend des Resultats des Anfrageterms zugewiesen werden können (z.B.: var X -> title {{ }}). Label Variablen gleichen Variabeln ohne Restriktionen. Sie stehen jedoch an der Position eines Labels (z.B.: student {{ var x {'Test'}}}). Die letzte Variablenart sind die Namespace-Variablen. Sie stehen an der Position von Namensräumen und

können nur an Namensräumen gebunden werden. Das Beispiel 15 zeigt, wie ein Anfrageterm mit den Variablen `Name` und `MatrNr` aufgebaut ist, der die Namen und falls vorhanden die Matrikelnummern von Studenten selektiert.

Beispiel 15 Verwendung von Variablen ohne Einschränkung in einem Anfrageterm

```
students {{
  student {{
    name { var Name },
    optional matrnr { var MatrNr }
  }}
}}
```

Anfrageterme können bezüglich der Breite sowie der Tiefe unvollständige formuliert werden. Bezüglich der Breite vollständig formulierte Terme enthalten alle Unterausdrücke. Vollständig formulierter Terme werden über einzelne eckige beziehungsweise geschweifte Klammern beschrieben. Unvollständige Anfragen werden durch doppelte Klammern([[,{{) beschrieben. Zusätzliche oder nicht spezifizierte Subterme reduzieren die Treffermenge in diesem Fall nicht.

Die zweite Dimension, in der Anfrageterme unvollständig definiert werden können, betrifft die Tiefe. Ähnlich dem aus XPath bekannten Konstrukt "//" (vgl. Kap. 3.1) bietet Xcerpt das Konstrukt `desc`(descendant). Der Ausdruck desc t ist für alle Datenterme erfüllt, die den Subterme t enthalten. Damit ist die Anfrage in Beispiel 16 erfüllt für alle Titelterme mit dem Titel „Data Terms“ in beliebiger Tiefe unterhalb des section Datenterms. Darüber hinaus dürfen, auf Grund der durch doppelte geschweifte Klammern ({{, }})unvollständig definierten Anfrage, alle Datenterme weitere Subterme besitzen.

Beispiel 16 Formulierung einer in der Breite und Tiefe unvollständig definierten Anfrage

```
report {{
  desc section {{
   title {{ "Data Terms" }},
  }}
}}
```

Graph Simulation Auch die Anfrageterme beziehungsweise Anfragemuster beschreiben, wie die Datenterme, Graphen. Mit Hilfe der sogenannten *Graph Simulation* werden alle Variablenbelegungen des Anfrageausdrucks gesucht. Dazu wird geprüft, ob der Anfragegraph ein Teilgraph des aus den Datentermen abgeleiteten Graphens ist. Eine Graph Simulation von einem Graphen G1=(V1, E1) nach G2=(V2, E2) ist eine Relation R zwischen ihren Knoten. Sie gilt, wenn für alle Knoten, die in R in Beziehung zueinander stehen ((v1, v2) in R) und eine

Kante a (v1, a, y1) in G1 existiert, auch eine Kante (v2, a, y2) in G2 existieren muss und so die Knoten y1 und y2 ebenfalls in R enthalten sind. Für die Beispielgraphen in Abbildung 9 bedeutet das, dass alle Knoten gleichen Namens in R enthalten sind. Die Knoten A der beiden Graphen stehen somit in Beziehung. Für die Simulation muss weiter gelten, dass jeder von A ausgehenden Kante in G1 eine entsprechenden Kante im Graph G2 gegenübersteht. Im Beispiel gilt dies nur für die Kante zwischen den Knoten A und B. Im nächsten Schritt müssen den ausgehenden Kanten des Knotens B in G1 Kanten in G2 gegenüberstehen. Für die Kanten zwischen den Knoten B und D sowie B und E ist diese Bedingung erfüllt.

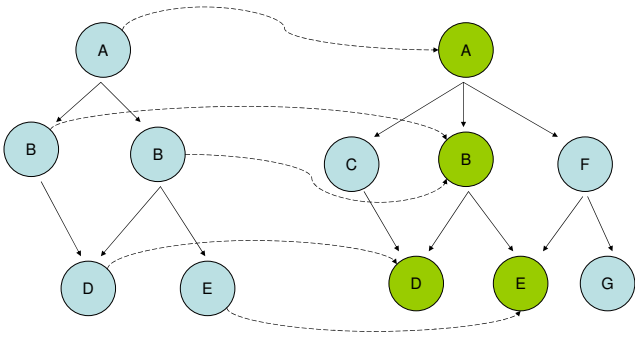

Abbildung 9. Beispiel einer Graph Simulation

Anfragen Die im vorigen Kapitel eingeführten Anfrageterme können durch AND- und OR-Operatoren verknüpft werden. Eine Anfrage besteht aus einem Anfrageterm oder der Verknüpfung von beliebig vielen Anfragetermen. Jeder Anfrageterm der Anfrage kann sich auf unterschiedliche Ressourcen beziehen. Das Beispiel 17 zeigt, neben der Fähigkeit mehrere Resourcen (bib.xml und reviews.xml) zu durchsuchen, auch, wie mit Hilfe des AND-Operators ein JOIN implementiert werden kann. Die gemeinsame Variable T legt den Titel als Join-Attribut fest. Die Anfrage wählt die Preise aller Bücher beziehungweise Einträge aus, die in beiden Quelldokumenten unter dem gleichen Titel zu finden sind. Weiterhin bietet Xcerpt einen not-Operator sowie einen Konditionaloperator(WHERE), welcher der WHERE-Klausel in SQL ähnelt. Diese sollen an dieser Stelle jedoch nicht vertieft werden [32].

Beispiel 17 Beispiel zu mehreren Ressourcen in einer Anfrage und der Implementierung eines Joins

```
and {
  in {
   resource [ "file:bib.xml" ],
   bib [[
    book [[
      title [ var T ],
      price [ var Pa ]
    ]]
   ]]
  },
  in {
   resource [ "file:reviews.xml" ],
   reviews [[
    entry [[
      title [ var T ],
      price [ var Pb ]
    ]]
   ]]
  }
}
```

Konstruktionsterme Mit Hilfe der Konstruktionsterme werden Variablen, die in den Anfragen gebunden wurden neu angeordnet und zu vollständigen Datentermen zusammengesetzt. Die Restrukturierung darf geordnete([) wie auch ungeordnete({) jedoch keine unvollständigen ([[, {{) Terme enthalten. Angenohmen eine Anfrage liefert die in Beispiel 18 angegebenen Bindungen für die Variablen `Titel` und `Author`.

Beispiel 18 mögliche Antwortsubstitutionen der Variablen Titel und Name einer Anfrage [32]

```
1.  Title title { "Vikinga Blot" }
           Author author { last { "Ingelman-Sundberg" }, first { "Catharina" }}

2.  Title title { "Folket i Birka på Vikingarnas Tid" }
           Author author { last { "Wahl" }, first { "Mats" }}

3.  Title title { "Folket i Birka på Vikingarnas Tid" }
           Author author { last { "Nordqvist" }, first { "Sven" }}
```

Der Konstruktionsterm in Beispiel 19 sammelt alle Substitutionen der Titel/Author Paare und fügt sie in results-Termen zusammen.

Beispiel 19 Beispiel für einen Konstruktionsterm

```
results{
  result{var title, var author}
}
```

liefert:

```
results {
  result {
    title { "Vikinga Blot" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  }
}

results {
  result {
    title { "Folket i Birka pâ Vikingarnas Tid" },
    author { last { "Wahl" }, first { "Mats" } }
  }
}

results {
  result {
    title { "Folket i Birka pâ Vikingarnas Tid" },
    author { last { "Nordqvist" }, first { "Sven" } }
  },
}
```

Gruppierung

Häufig ist es wichtig alle Substitutionen der Variablen in einem einzigen Term zu erhalten. Mit Hilfe des Schlüsselworts `all` gruppiert man über alle möglichen Subterme verschiedener Variablenbelegungen. Das Schlüsselwort `some` schränkt die Anzahl berücksichtigter Terme ein.

Möchte man in Anlehnung an das vorige Beispiel nun alle Titel - Author Substitutionen unterhalb eines *results*-Ausdrucks erhalten, muss der Konstruktionsterm um das Schlüsselwort `all` erweitert werden (vgl. Beispiel 20).

Beispiel 20 Einsatz des Gruppierungsoperator `all`

```
results{
all result{var title, var author}
}
```

liefert:

```
results {
  result {
    title { "Vikinga Blot" },
    author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
  }
  result {
    title { "Folket i Birka på Vikingarnas Tid" },
    author { last { "Wahl" }, first { "Mats" } }
  }
  result {
    title { "Folket i Birka â Vikingarnas Tid" },
    author { last { "Nordqvist" }, first { "Sven" } }
  },
}
```

Selbstverständlich können die beiden Operatoren `all` und `some` auch auf Subterme angewendet werden. Eine *freie Variable* ist nicht durch einen der Operatoren erfasst, steht also nicht unterhalb eines Ausdrucks, der einem `all` oder `some` folgt. Zu jeder freien Variable werden alle Variablenbindungen der gebundenen Variable gesucht. In Beispiel 21 ist die Variable `title` frei und die Variable `author` gebunden. Zu jedem Titel werden somit alle Authoren gesucht und in einem `result`-Ausdruck zusammengefügt.

Beispiel 21 Einsatz gebundener und freier Variablen

```
result{var title, all var author}
```

liefert:

```
result {
  title { "Vikinga Blot" },
  author { last { "Ingelman-Sundberg" }, first { "Catharina" } }
}

result {
  title { "Folket i Birka på Vikingarnas Tid" },
  author { last { "Wahl" }, first { "Mats" },
  author { last { "Nordqvist" }, first { "Sven" }
}
```

Xcerpt bietet weitere Möglichkeiten Daten zu gruppieren, wie die Verschachtelung, den expliziten Einsatz von `group by`-Klauseln und das Sortieren von Ergebnissen. Auf eine detaillierte Einführung dieser Konzepte wird an dieser Stelle verzichtet.

Wie viele andere Sprachen bietet auch Xcerpt eine Auswahl von Funktionen und Aggregatfunktionen an. Im Vergleich zu Funktionen haben Aggregatofunktionen keine feste Anzahl von Argumenten. In [32] befindet sich eine Übersicht

über die von Xcerpt bereitgestellten Standardfunktionen. Der Umfang der angebotenen Funktionen und Aggregatfunktionen ist im Vergleich zu XPath (vgl. Kap. 3.1) gering, stand jedoch auch nicht im Mittelpunkt der Entwicklung der Sprache, die bisher nur in einer rudimentären Implementierung vorliegt [32].

Construct-Query-Rule Nachdem nun alle Abschnitte eines Xcerpt Programms unabhängig von einander vorgestellt worden sind, bleibt abschließend noch einzuführen, wie sie kombiniert werden können. Ein Programm entspricht folgendem Grundmuster:

```
CONSTRUCT
 <construct term>
FROM
 <query term>
END
```

Das Beispiel 22 zeigt ein vollständiges Anfrageprogramm. Dabei ist zu beachten, dass bereits in diesem einfachen Beispiel zwei unterschiedliche Quelldokumenten (bib.xml, reviews.xml) mit verschiedenen Schemastrukturen ausgewertet werden. Das Schema des *bib.xml*-Dokuments beschreibt Buchtitel unterhalb eines `book`-Elements, welches Teil des `bib`-Ausdrucks ist. In der zweiten Quelle werden die Titel unterhalb eines `entry`-Ausdrucks geführt, der wiederum Teil des `reviews`-Ausdrucks ist. Ziel der Anfrage ist, die Preise aller Bücher aufzulisten, die in beiden Quellen enthalten sind.

Beispiel 22 Beispiel eines vollständigen Anfrageprogramms

```
CONSTRUCT
  books-with-prices [
    all book-with-prices [
      title [ var T ], price-a [ var Pa ], price-b [ var Pb ]
    ]
  ]
FROM
  and {
    in {
      resource [ "file:bib.xml" ],
      bib [[
        book [[
          title [ var T ], price [ var Pa ]
        ]]
      ]]
    },
    in {
      resource [ "file:reviews.xml" ],
      reviews [[
        entry [[
```

```
          title [ var T ], price [ var Pb ]
        ]]
      ]]
    }
  }
END
```

Regelverkettung Eine wichtige Eigenschaft von Xcerpt ist die Fähigkeit, Regeln zu verketten. Regelverkettung bietet den Vorteil, dass Anfragen in übersichtlichere Einzelteile zerlegt werden können. Keine der früheren Sprachen XML-QL oder UnQL boten diese Möglichkeit [32].

Beispiel 23 Beispiel zur Regelverkettung

```
CONSTRUCT
  table [
   tr [ td [ "Title" ], td [ "Price at A" ], td [ "Price at B" ] ],
   all tr [ td [ var Title], td [ var PriceA], td [ var PriceB] ]
  ]
FROM
  books-with-prices [[
    book-with-prices [[
      title [[ var Title ]],
      price-a [[ var PriceA]],
      price-b [[ var PriceB]]
    ]]
  ]]
END
```

Das Beispiel 23 nutzt die Ergebnisse der vorigen Beispiels (vgl. Bsp. 22) und gibt die Daten in Form einer HTML-Tabelle aus.

Zusammenfassung Die Entwicklung der XML-Anfragesprache Xcerpt ist geprägt von der Anforderung, verschiedene Quellen semistrukturierter Daten, die in verschiedenen Metamodellen und Schemata vorliegen, verarbeiten zu können. Als Grundlage wurde dazu ein musterbasierter Ansatz gewählt. Die Sprache gliedert sich in die drei Termvarianten: Daten-, Anfrage- und Konstruktionsterme, die hier vorgestellt wurden.

4 Zusammenfassung

In dieser Arbeit wurden die verschiedenen XML-Parserkonzepte am Beispiel der SAX-, DOM- und StAX-APIs vorgestellt. Sie unterstreichen wie unterschiedlich die Anwendungsfälle und damit auch die Anforderungen an die XML-Verarbeitung

sind. In datenbankorientierten Anwendungen von XML sind Anfragesprachen von zentraler Bedeutung. Für jedes der beidengrundlegenden Sprachkonzepte, den navigierenden beziehungsweise musterbasierten Sprachen wurden mit XQuery/XPath beziehungsweise Xcerpt jeweils ein Beispiel vorgestellt. Durch die Weiterentwicklung von XML und XML-nahen Standards wie RDF, der Einführung neuer Standards und der Entstehung neuer Einsatzmöglichkeiten von XML werden sich die Sprachen auch in Zukunft weiterentwickeln. Xcerpt als jüngste Entwicklung, der hier vorgestellten Sprachen, bietet die breiteste Unterstützung verschiedener Technologien an. Die Sprachen, die in Zukunft von Relevanz sein sollen, müssen für die verschiedenen Einsatzmöglichkeiten von XML geeignet sein. Dazu müssen sie eine gute Integration neuer Technologien und eine einheitliche Verwendung verschiedenen Technologien anbieten.

Literatur

[1] Extensible Markup Language (XML) 1.1 http://www.edition-w3c.de/TR/2004/REC-xml11-20040204/

[2] Dodds, L.:Investigating the Infoset 2000 http://www.xml.com/pub/a/2000/08/02/deviant/infoset.html

[3] Wittenbrink, H; Bergmann O.: XML - Grundlagen Teia Lehrbuch Verlag, 2005

[4] Schild, K.: Verarbeitung von XML-Dokumenten 2003 http://www.xml-clearinghouse.de/ch-veranstaltungen/SchulungKS/xml-dom.htm

[5] Webseite des SAX Projekts http://www.saxproject.org

[6] Document Object Model (DOM) Level 1 Specification Version 1.0 http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

[7] Slomiski A.: On Using XML Pull Parsing Java APIs 2004 http://www.xmlpull.org/history/index.html

[8] Harold, Elliotte R.: An Introduction to StAX 2003 http://www.xml.com/pub/a/2003/09/17/stax.html?page=1

[9] Nehrer, P.: StAX'ing up XML, Part 2: Pull parsing and events 2006 http://www.ibm.com/developerworks/xml/library/x-stax2.html

[10] Ullenboom, C.: Java ist auch eine Insel, 6. Auflage Galileo Computing 2007

[11] Opletal, Sascha: Einsatz einer XML-Anfragesprache für Transformationszwecke Universität Stuttgart

[12] W3 Recommendation zum XPath Datenmodell: http://www.w3.org/TR/xpathdatamodel/NamespaceNode

[13] Abiteboul, S.; Quass, D.; McHugh, J.; Widim, J.; Wiener, J.: The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries, 1(1):68-88, April 1997 http://infolab.stanford.edu/lore/pubs/lorel96.pdf

[14] Deutsch, A.; Fernandez, M.; Florescu, D.; Levy, A.; Suciu, D.: XML-QL: A Query Language for XML 1998 http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

[15] Robie, J.; Lapp, J.; Schach, D.: XML Query Language (XQL) 1998 http://www.w3.org/TandS/QL/QL98/pp/xql.html

[16] Ceri, S.; Comani, S.; Damiani, E.; Fraternali, P.; Paraboschi, S.; Tanca, L.: XML-GL: a Graphical Language for Querying and Restructuring XML Documents http://www8.org/w8-papers/1c-xml/xml-gl/xml-gl.html

[17] XML Pointer Language (XPointer) Version 1.0 http://www.w3.org/TR/WD-xptr

[18] XSL Transformations (XSLT) Version 1.0 W3C Recommendation 1999 http://www.w3.org/TR/xslt

[19] Übersetzung der W3 Recommendation zu XPath 1.0: http://www2.informatik.hu-berlin.de/~obecker/obqo/w3c-trans/xpath-de

[20] Lenz, E., What ist new in XPath 2.0, 20.03.2002: http://www.xml.com/pub/a/2002/03/20/xpath2.html?page=1

[21] Wilde, E.: XML XPath Language(XPath) UCB iSchool 19.09.2006 http://dret.net/lectures/xml-fall06/xpath-chapter.pdf

[22] W3 Recommendation zu XPath http://www.w3.org/TR/xpath20/

[23] Chamberlin D.: XQuery: An XML query language IBM SYSTEMS JOURNAL, Vol 41, No 4, 2002

[24] Pandrangi, S.;Cheng, A.;Zhang, H.;Xu, Q.; Gan, J.: An Introduction to XQuery 2002 http://www.devx.com/xml/Article/9816

[25] XQuery 1.0 and XPath 2.0 Functions and Operators 23.01.2007 http://www.w3.org/TR/2007/REC-xpath-functions-20070123/

[26] Kepser, S.: A Simple Proof for the Turing-Completeness of XSLT and XQuery 2004 http://tcl.sfs.uni-tuebingen.de/~kepser/papers/EML2004Kepser01.pdf
[27] http://www.idealliance.org/proceedings/xtech05/papers/02-03-01/
[28] Cohen, F.: Debunking XQuery myths and misunderstandings 2005 http://www-128.ibm.com/developerworks/xml/library/x-xqmyth.html
[29] Laux, A.; martin, L.: XUpdate Working Draft 2000 http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html
[30] XQuery Update Facility 1.0 W3C Working Draft 2007 http://www.w3.org/TR/2007/WD-xquery-update-10-20070828/
[31] Bry, F.; Furche, T.; Linse, B.: Let's Mix It: Versatile Access to Web Data in Xcerpt 2006
[32] Schaffert, S.: Xcerpt: A Rule-Based Query and Transformation Language for the Web LMU München Okt. 2004